Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Mateja Mencin

Vsebinsko zavedno spreminjanje velikosti slik v OpenCL

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: izr. prof. dr. Uroš Lotrič

Ljubljana, 2016

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Vsebinsko zavedno spreminjanje velikosti slik v OpenCL

Tematika naloge:

Pomemben cilj pri razvoju ogrodja OpenCL je bil v programski podpori najrazličnejše strojne opreme, od procesorjev, grafičnih kartic do vezij FPGA in drugih naprav. V okviru ogrodja izdelajte kar se da splošen program za vsebinsko zavedno spreminjanje velikosti slik po metodi rezanja šivov. Za posamezne korake metode preverite različne možnosti paralelizacije v ogrodju OpenCL. Na izbrani strojni opremi rezultate izvajanja paralelnega programa primerjalno ovrednotite s programom, ki teče izključno na centralni procesni enoti.

Izjava o avtorstvu diplomskega dela

Spodaj podpisana Mateja Mencin, z vpisno številko 63100033, sem avtorica diplomskega dela z naslovom:

Vsebinsko zavedno spreminjanje velikosti slik v OpenCL (angl. *Content-aware image resizing in OpenCL*)

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelala samostojno pod mentorstvom izr. prof. dr. Uroša Lotriča
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 28. februarja 2016 Podpis avtorja:

Na tem mestu bi se zahvalila svojemu mentorju izr. prof. dr. Urošu Lotriču za potrpežljivost in pomoč pri izdelavi diplomske naloge.

Staršem, za podporo v vseh letih študija.

Kazalo

Seznam uporabljenih kratic

kratica	angleško
GPU	graphics processing unit
CPU	central processing unit
SMX	next generation streaming multiprocessor
GPC	graphics processing clusters
GPGPU	general purpose graphics processing unit
OpenCL	Open Computing Language

kratica	slovensko
GPE	grafična procesna enota
CPE	centralna procesna enota
SMX	multiprocesor naslednje generacije
GPC	grafična procesna skupina
GPGPU	grafične procesne enote za splošne namene
OpenCL	odprt računski jezik

Povzetek

Naslov: Vsebinsko zavedno spreminjanje velikosti slik v OpenCL

Namen diplomskega dela je preveriti ali se izbrani algoritem za vsebinsko zavedno spreminjanje velikosti slik, dejansko izvede hitreje na grafično procesnih enotah v primerjavi z izvedbo na centralnih procesnih enotah. Za ta namen smo implementirali algoritem rezanja šivov, ki je postopek za spreminjanje dimenzij slike z upoštevanjem vsebine same slike.

Pri algoritmu rezanja šivov dimenzijo slike spremenimo tako, da odrežemo ali dodamo šiv, ki je povezana pot iz ene strani slike na drugo. Šiv predstavlja najmanj pomemben del slike in ga zato lahko odrežemo ali dodamo, ne da bi s tem izgubili pomemben del slike. Pri testiranju smo ugotovili, da je algoritem uspešen na slikah z monotonim ozadjem. Ker algoritem ni bil glavni del diplomske naloge, se z izboljšavo le-tega nismo ukvarjali.

Za implementacijo omenjenega algoritma na grafično procesnih enotah (ang. graphical processing unit, GPU) smo uporabili heterogeno programsko ogrodje OpenCL. OpenCL je standard za heterogeno paralelno računaje na različnih platformah različnih proizvajalcev. Njegovo arhitekturo lahko razdelimo na platformni, izvajalni, pomnilniški in programski model. Vse to je podrobneje opisano v tretjem poglavju.

Četrto poglavje zajema opis naše implementacije algoritma rezanja šivov. Tega smo se lotili na dva načina. Prvi je rezanje enega šiva, pri katerem za vsak šiv, ki ga želimo odrezati, ponovno poračunamo energijo slike. Drugi pa je rezanje več šivov naenkrat. V tem primeru poizkušamo poiskati več šivov, ki jih na podlagi poračunane energije lahko odrežemo. Postopek ponavljamo

dokler ne pridemo do želenih dimenzij.

Ugotovili smo, da lahko izbira delovne skupine močno vpliva na čas izvajanja ščepca. Ravno tako na hitrost izvedbe močno vpliva tudi implementacija samega ščepca. Nepravilen pristop lahko močno upočasni njegovo izvajanje. To se lepo vidi pri implementaciji rezanja več šivov naenkrat. V tem primeru je bilo izvajanje algoritma hitrejše na centralno procesni enoti(angl. central processing unit, CPU), kot pa na GPE. Pri načinu rezanja samo enega šiva pa smo algoritem uspešno implementirali, saj je bila pohitritev velika.

Ključne besede: GPE, CPE, OpenCL, algoritem rezanja šivov.

Abstract

Title: Content-aware image resizing in OpenCL

The purpose of this thesis was to test if the algorithm for content-aware image resizing runs faster on graphics processing unit in comparison to central processing unit. For that we chose content-aware image resizing algorithm called seam carving.

With seam carving we can change image dimensions by finding the optimal seam which we can carve out or put in, depending on weather we want to shrink or enlarge the image. Seam is connected path from one side of the image to another and holds least important information of the image. With our testing we realized that this algorithm works best in images with monotone background. Because algorithm itself was not the purpose of this thesis we did not try to improve it.

For implementation of this algorithm on graphics processing unit we used heterogeneous programming framework called OpenCL. OpenCL is a standard for heterogeneous parallel computing on cross-vendor and cross-platform hardware. We can describe OpenCL architecture with platform model, execution model, memory model and programming model. Each of them is described in details in chapter three.

In chapter four we look at our implementation of seam carving algorithm. We had two approaches. One is carving one seam at the time, which means recalculating energy and its cumulative every time we carve out a seam. Second approach is carving multiple seams at a time. In this case we try to find more seams that we can carve out based on calculated energy and

cumulative. We repeat the process until we get the desired image dimensions.

Based on testing we realised that choosing the right work group size is really important, as well as implementation of kernels. If we choose wrong approach we can slow down its execution considerably, which is evident from the results of second approach. In this case the execution of the algorithm on central processing unit was faster then execution of it on graphics processing unit. We were more successful with implementation of first approach which runs faster on graphics processing unit then on central processing unit.

Keywords: GPU, CPU, OpenCL, seam carving algorithm.

Poglavje 1

Uvod

V zadnjem času se količina podatkov hitro povečuje, s tem pa narašča tudi potreba po njihovi hitrejši obdelavi. V porastu je uporaba grafičnih procesnih enot (ang. graphical processing unit, GPU) za računanje, ki nima nobene povezave z izrisom grafike. Tako so grafične procesne enote postale procesorji za splošne namene (angl. general purpose graphics processing unit, GPGPU) in ne samo procesorji za grafične rutine.

Namen diplomske naloge je preizkusiti kolikšno pohitritev lahko dosežemo pri izvajanju določenega algoritma na grafično procesni enoti v primerjavi z izvajanjem na centralni procesni enoti (ang. central processing unit, CPU). V ta namen smo izbrali ogrodje OpenCL, ki podpira najrazličnejšo strojno opremo, kot so grafične kartice, procesorji in druge naprave. To pa pomeni, da lahko program prenesemo na katero koli napravo, ki podpira OpenCL, kar je ena izmed prednosti tega ogrodja. Za testiranje smo izbrali algoritem rezanja šivov, ki je postopek za vsebinsko zavedno spreminjanje velikosti slike.

Najprej si bomo pogledali zakaj je algoritem, ki pri spreminjanju velikosti upošteva vsebino, boljši od standardnih metod za spreminjanje velikosti slik kot so raztegovanje ali krčenje ter rezanje. Opisali bomo postopek s katerim poiščemo najbolj optimalen šiv. Postopek zajema računanje energije slike, ter na podlagi tega še računanje komulative slike, iskanje šiva in nato rezanje

le-tega.

Za tem si bomo pogledali kako z uporabo ogrodja OpenCL implementiramo kodo, s katero poiščemo platformo in napravo na kateri želimo izvajati naše računanje, ter poskrbi za izvajanje ščepcev (angl. kernel) na izbrani napravi. Videli bomo tudi, da je OpenCL razdeljen na platformni, izvajalni, pomnilniški in programski model, ter vsakega posebej podrobneje opisali. Opisali bomo tudi posamezne ščepce in njihov namen. Vzporedno pa bomo implementirali tudi kodo, ki se izvaja izključno na CPE, za kasnejšo časovno primerjavo med obema.

Na koncu si bomo podrobneje pogledali koliko časa je potrebnega za prenos podatkov na grafično kartico, glede na velikost teh podatkov. Primerjali bomo posamezne časovne meritve funkcij na CPE in njihovih različic na GPE, ter na koncu še čase celotnega programa.

Poglavje 2

Rezanje šivov

V tem poglavju si bomo najprej pogledali kaj je rezanje šivov in kje se uporablja. Definirali bomo šiv in pojem sosednosti. Za tem si bomo pogledali postopek izračuna šiva, ki zajema poračun energije in komulative slike, ter na koncu iskanje optimalnega šiva in rezanje le-tega.

V današnjem času uporabljamo za ogledovanje digitalnih slik mnogo različnih naprav, ki imajo različne dimenzije zaslona. Postopki prilagajanja slike zaslonu naprave vključujejo raztegovanje oziroma krčenje in rezanje. Kot vidimo na sliki 2.1, standardno raztegovanje slike ni zadostno, saj ne upošteva vsebine slike. Običajno se uporablja le enotno raztegovanje velikosti, ker neenotno raztegovanje popači sliko. Rezanje je ravno tako omejeno, saj se lahko odstrani le piksle na robu slike. Učinkovitejše spreminjanje velikosti je mogoče doseči le z upoštevanjem vsebine slike in ne samo z upoštevanjem geometrijske omejitve. S postopkom za spreminjanje velikosti slike, imenovanim rezanje šivov, lahko spremenimo velikost slike bolj elegantno, saj poreže ali vstavi piksle v različnih delih slike [3]. Primerjava načinov je prikazana na sliki 2.1.

(a) vhodna slika [7]

(b) raztegovanje

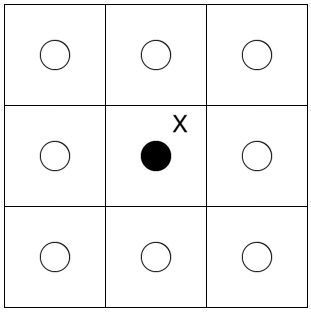
(c) rezanje

(d) rezanje šivov

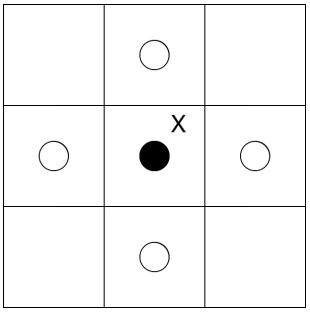
Slika 2.1: Primerjava različnih načinov spreminjanja velikosti slike. Slika (a) prikazuje originalno sliko, ki jo želimo zmanjšati po dolžini tako, da bosta dolžina in širina enaki. Na sliki (b) vidimo rezultat pri uporabi raztezanja oziroma, v tem primeru krčenja slike - vidimo da je slika popačena. Slika (c) prikazuje rezultat rezanja slike - slika sicer ni popačena, vendar smo izgubili del slike kjer je oseba. Slika (d) pa prikazuje rezultat po uporabi algoritma za rezanje šivov - tu vidimo, da so se ohranili vsi pomembni elementi slike in so bili odstranjeni samo nepomembni piksli.

Šiv je optimalna 8-sosedna povezana pot pikslov na eni sliki od zgoraj navzdol ali od leve proti desni. Optimalnost je opredeljena s funkcijo izračuna energije slike. Z zaporednim odstranjevanjem ali vstavljanjem šivov lahko zmanjšamo ali povečamo velikost slike v obeh smereh.

Algoritem rezanja šivov omogoča spremembo velikosti slike s spreminjanjem najmanj opaznih pikslov na sliki. Najbolj razširjena uporaba algoritma za rezanje šivov je zmanjšanje velikosti slike le po eni dimenziji. To je mogoče



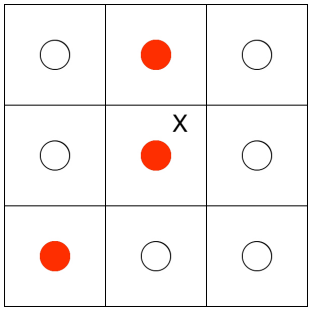
(a) 8-sosednost



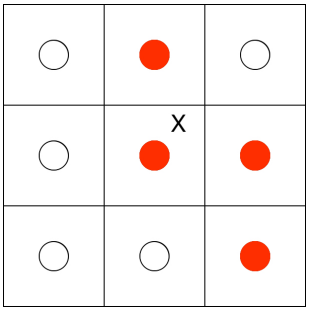
(b) 4-sosednost

Slika 2.2: Prikaz 8-sosednosti (a), kjer ima piksel X 8 sosedov (2 po horizontali, 2 po vertikali in 4 po diagonali) in 4-sosednosti (b), kjer ima piksel X 4 sosede (2 po horizontali in 2 po vertikali).

doseči z iskanjem in nato z odstranjevanjem en piksel širokega šiva na sliki. Če so piksli na šivu podobni okoliškim pikslom, potem je lahko njihova odstranitev neopažena. Obstajajo tudi druge uporabe algoritma, ki vključujejo povečanje velikosti slike, spreminjanje velikosti slike v dveh dimenzijah in celo odstranitev določenega objekta na sliki, z uporabo algoritma rezanja šivov.



(a) pravilen šiv



(b) napačen šiv

Slika 2.3: Za pravilen šiv je v vsaki vrstici izbrana samo ena točka oziroma piksel. Slika (a) prikazuje pravilen šiv, slika (b) pa nepravilen šiv.

2.1 Postopek izračuna šiva

Šiv se poračuna po naslednjem postopku:

- **Izračun energije:** Prvi korak je izračun energije vsakega piksla, ki je mera pomembnosti posameznega piksla. Višja je energije, manj verjetno je, da bo piksel del šiva.
- **Izračun komulative:** Poračunamo glede na energijo, ki smo jo dobili v prejšnjem koraku. Komulativna vrednost piksla je njegova energijska vrednost, ki ji prištejemo še minimalno komulativno vrednost najmanjšega spodnjega soseda.
- **Iskanje in rezanje šiva:** Poiščemo najbolj optimalen šiv in ga izrežemo.

2.1.1 Izračun energije

Algoritem rezanja šiva podpira več vrst energijskih funkcij kot so gradient magnitude, entropija, sledenje premikom oči in drugi. V tej nalogi smo uporabili Sobelov operator za izračun energije slike.

Sobelov operator uporablja dve matriki, s katerima opišemo konvolucijsko masko in izvajamo konvulacijo nad originalno sliko. Dimenzije matrike so ponavadi lihe, tako da je sredina preprosto določljiva. V tej nalogi smo uporabili matrike velikosti 3x3. Eno matriko uporabimo za detekcijo sprememb v horizontalni smeri, drugo pa za detekcijo sprememb v vertikalni smeri. Formulo zapišemo kot:

$$\mathbf{H}_x = \begin{bmatrix} \mathbf{-1} & \mathbf{0} & \mathbf{1} \\ \mathbf{-2} & \mathbf{0} & \mathbf{2} \\ \mathbf{-1} & \mathbf{0} & \mathbf{1} \end{bmatrix} * \mathbf{A} \text{ in } \mathbf{H}_y = \begin{bmatrix} \mathbf{-1} & \mathbf{-2} & \mathbf{-1} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{1} & \mathbf{2} & \mathbf{1} \end{bmatrix} * \mathbf{A} \tag{2.1}$$

Tu matrika $\mathbf{H}_x$ predstavlja rezultate sprememb v horizontalni smeri in matrika $\mathbf{H}_y$ rezultate sprememb v vertikalni smeri, * pa predstavlja konvulacijo nad originalno sliko $\mathbf{A}$, ki je ravno tako zapisana v obliki matrike. Pri obdelavi premikamo konvolucijsko masko po sliki tako, da je obravnavani

piksel na sredini. Ne glede na to ali je slika barvna ali črno bela, se po sliki premikamo za en piksel, razlika je samo v tem, da pri barvni sliki poračunamo vsako barvo modela, s katerim je slika predstavljena, posebej.

Da dobimo energijo slike, lahko rezultata $\mathbf{H}_x$ in $\mathbf{H}_y$ združimo po sledeči formuli:

$$\mathbf{H} = \sqrt{\mathbf{H}_x^2 + \mathbf{H}_y^2} \tag{2.2}$$

Slika 2.4: Originalna vhodna slika [8].

Slika 2.5: Prikaz poračuna energije na sliki.

2.1.2 Izračun komulative

Poračun komulative za vertikalen šiv lahko začnemo v zgornji ali spodnji vrstici. V našem primeru smo začeli v spodnji. Ker ta nima naslednje vrstice, se energija posameznega piksla samo prepiše v tabelo komulativ, kar zapišemo kot:

$$C(v, j) = H(v, j), \tag{2.3}$$

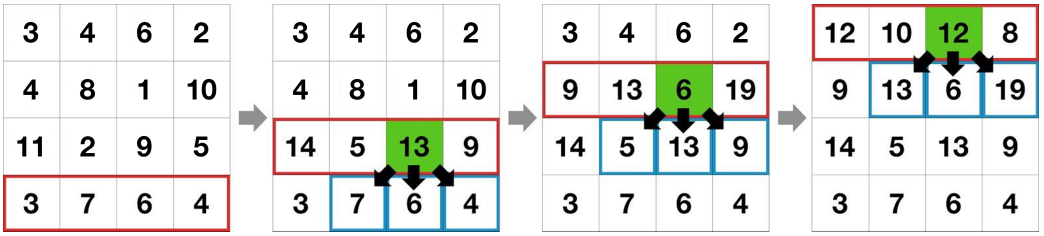
kjer *C(v,j)* predstavlja komulativo in *H(v,j)* energijo piksla. Pri tem je v enak višini slike, j pa ima vrednost med 1 in dolžino slike. Nato se premaknemo vrstico višje, kjer je komulativa piksla enaka vrednosti pripadajoče energije piksla, kateri dodamo minimalno vrednost energij treh sosednjih pikslov v spodnji vrstici. Postopek ponavljamo dokler ne pridemo do prve vrstice. Slika 2.7 prikazuje izračun komulative. Zapis poračuna za posamezen piksel je naslednji:

$$C(i, j) = H(i, j) + \min(H(i-1, j-1), H(i-1, j), H(i-1, j+1)), \tag{2.4}$$

kjer ima i vrednost med 1 in v-1.

3	**4**	**6**	**2**
4	**8**	**1**	**10**
11	**2**	**9**	**5**
3	**7**	**6**	**4**

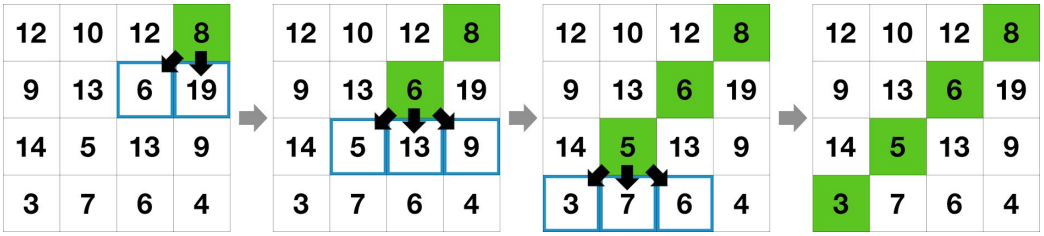
Slika 2.6: Tabela s poračunanimi energijami.



Slika 2.7: Za prikaz postopka izračuna komulativ smo vzeli tabelo s poračunanimi energijami na sliki 2.6. V spodnji vrstici samo prepišemo energijske vrednosti pikslov. Za poračun zelenega piksla v vrstici višje smo vzeli njegovo energijsko vrednost 9 in ji prišteli minimum(7, 6, 4) spodnjih sosedov. Končna vrednost je $9 + 4 = 13$. Tako poračunamo za vsak piksel posebej.

2.1.3 Iskanje šiva

Šiv poiščemo na podlagi poračunanih minimalnih komulativ. V našem primeru se postavimo v prvo vrstico in poiščemo minimalno vrednost komulative in ta piksel označimo za del našega šiva. Potem poiščemo minimalno vrednost komulativ spodnjih sosedov in ta piksel dodamo v šiv, postopek ponavljamo dokler ne pridemo do zadnje vrstice. Potem nam preostane samo še to, da šiv izrežemo in s tem sliko zožamo za en piksel.



Slika 2.8: Iskanje šiva na poračunani komulativi. Začnemo v prvi vrstici, poiščemo najmanjši element in ga označimo kot del šiva. V našem primeru je to zadnji, 3. element. To je naše izhodišče za določitev šiva. V 2. vrstici poiščemo minimum sosedov zgoraj izbrane točke, ki je naslednji element šiva. Postopek ponavljamo dokler ne pridemo do zadnje vrstice. Naš šiv je {(0,3), (1,2), (2,1), (3,0)}, na sliki je zeleno obarvan.

Slika 2.9: Prikaz šiva na sliki.

Poglavje 3

OpenCL

V tem poglavju si bomo pogledali kaj je OpenCL in zakaj ga uporabljamo. Videli bomo, da OpenCL arhitekturo lahko razdelimo na platformni, izvajalni, pomnilniški in programski model, ter vsakega od njih podrobno opisali.

Vzporedno računanje je vrsta računanja v katerem se številni izračuni izvedejo istočasno, ob domnevi, da se velike računske probleme lahko razdeli na manjše, ki so nato rešeni istočasno, kar izredno pospeši izvajanje programa. Seveda pa mora biti istočasnost podprta, ne samo s strani programske opreme, ampak tudi s strani strojne opreme.

Oktobra 2010 je super računalnik Tianhe-1A na Kitajskem, postal eden izmed najbolj zmogljivih računalnikov na svetu. Bil je eden prvih super računalnikov, ki je uporabljal grafične procesne enote (v nadaljevanju GPE) za vzporedno računanje in s tem pohitril izvajanje programov. Tako so grafične procesne enote postale procesorji za splošne namene in ne samo procesorji za grafične rutine. Pred letom 2010 nihče ni namenjal velike pozornosti računanju na grafično procesnih enotah, dandanes pa inženirji in akademiki prihajajo do ugotovitve, da sistemi centralnih procesnih enot (v nadaljevanju CPE) v povezavi z GPE predstavljajo prihodnost superračunalnikov.[1]

Pojavi se pomembno vprašanje, kako programirati na teh novih sistemih. Odgovor je OpenCL (angl. Open Computing Language). OpenCL je heterogeno programsko ogrodje, ki ga upravlja neprofitni konzorcij Khronos

Group. Je standard za heterogeno paralelno računaje na različnih platformah različnih proizvajalcev. Zagotavlja abstraktnost na višjem nivoju za rutine strojne opreme na nižjem nivoju, kot tudi dosleden pomnilniški in izvajalni model za ravnanje z vzporedno izvajajočo kodo. Ravno ta abstraktnost na višjem nivoju je prednost ogrodja OpenCL, saj omogoča preprost prenos kode, brez popravkov, iz preprostih vgrajenih mikrokrmilnikov na Intelove ali AMD-jeve centralne procesne enote, kot tudi na grafične procesne enote[4].

Arhitekturo OpenCL lahko razdelimo na štiri dele, ki jih imenujemo modeli. To so:

- **Platformni model:** Določa, da je na gostitelju en procesor, ki koordinira izvedbo, ter na napravah, ki podpirajo OpenCL in so povezane z gostiteljem, eden ali več procesorjev za izvajanje kode. Opredeljuje abstraktni model strojne opreme, ki ga uporabljajo programerji, ko pišejo funkcije imenovane ščepci, ki se izvajajo na napravah.

- **Izvajalni model:** Določa kako je konfigurirano okolje OpenCL na gostitelju in kako se ščepci izvajajo na napravah. To vključuje pripravo OpenCL-ovega konteksta na gostitelju, zagotavljanje mehanizmov za interakcijo med gostiteljem in napravo, ter definira model sočasnosti za ščepce (angl. kernel), ki se izvajajo na napravah.

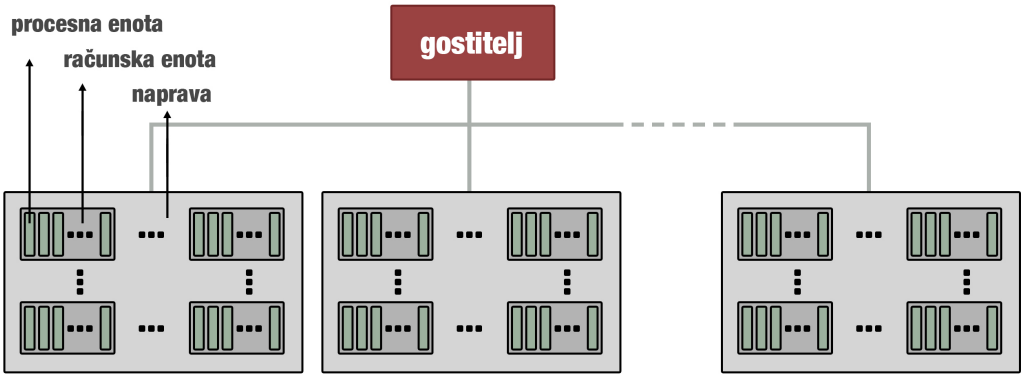
- **Pomnilniški model:** Definira abstraktno hierarhijo pomnilnika, ki ga ščepec uporablja, ne glede na dejansko pomnilniško arhitekturo na napravi. Ta model zelo spominja na trenutni pomnilnik na grafičnih karticah, vendar to ne omejuje uporabnosti ogrodja OpenCL na drugih napravah za pohitritev.

- **Programski model:** Definira, kako se model sočasnosti preslika v dejansko fizično strojno opremo.

3.1 Platformni model

Model je sestavljen iz številnih računskih naprav, ki podpirajo OpenCL in so povezane z gostiteljem. Vsaka naprava je razdeljena na eno ali več računskih enot, ki so funkcionalno neodvisne ena od druge in so naprej razdeljene na več procesnih enot. Dejansko računanje se izvaja na teh procesnih enotah.[5] Model je prikazan na sliki 3.1.

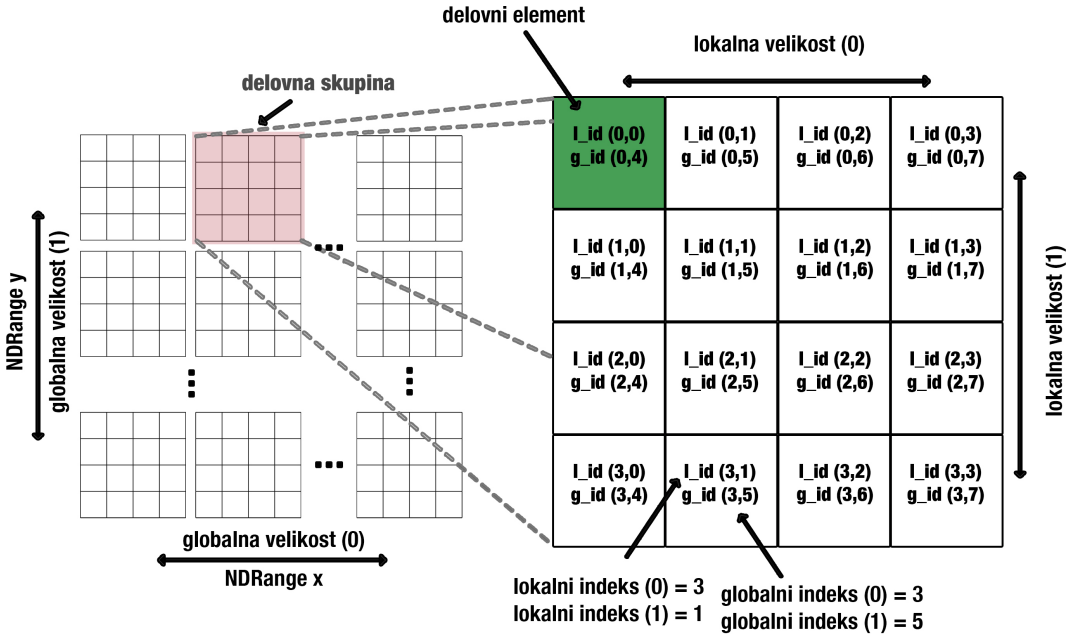
OpenCL definira tudi svoj jezik, ki je podoben programskemu jeziku C. Funkcije, ki se izvedejo na napravi se imenujejo ščepci. OpenCL-ov program je implementiran tako na strani gostitelja, kot na strani naprave. Tisti del kode, ki se izvede na gostitelju, pošlje ščepec preko ukazov na napravo. Naprava nato izvede računske ukaze na njenih procesnih enotah.[5] Ščepec se lahko izvaja vzporedno na vseh ali na večini procesnih enot. Kako je naprava razdeljena na računske enote in naprej na procesne enote, je odvisno od proizvajalca. OpenCL poleg programskega jezika definira tudi vmesnik uporabniškega programa. Ta omogoča programu, ki se izvaja na gostitelju, da zažene ščepce na napravi in upravlja z njenim pomnilnikom, ki je vsaj konceptualno ločen od pomnilnika na gostitelju.



Slika 3.1: Prikaz OpenCL-ove arhitekture z vidika platformnega modela. Kot vidimo so na gostitelja povezane računske naprave. Vsaka ta naprava je naprej razdeljena na eno ali več računskih enot, ki so naprej razdeljene na procesne enote, na katerih se izvaja računanje.

3.2 Izvajalni model

Izvajanje OpenCL programa izgleda tako, da se program zažene na gostitelju in ta potem pošlje ščepce v izvedbo na eno ali več naprav. Ko se ščepec pošlje v izvedbo, se določi indeksni prostor tako, da se sprožijo delovne enote za izvajanje vsake točke v tem prostoru. Vsaka delovna enota se identificira s svojim globalnim indeksom in izvede isto kodo, ki je v ščepcu. Delovne enote so združene v delovne skupine in tudi vsaka skupina ima svoj unikaten indeks. Indeksni prostor se imenuje NDRange in opiše N-dimenzionalni prostor, kjer je N velikosti od 1 do 3.[2] Vsaka delovna enota ima svoj globalni indeks in znotraj delovne skupine tudi svoj lokalni indeks. Ti indeksi so pridobljeni iz NDRanga. Isto velja za delovne skupine. Prikaz je na sliki 3.2.



Slika 3.2: Prikaz dvodimenzionalnega indeksnega prostora. Na sliki vidimo, da so lokalni indeksi vezani na velikost delovne skupine in so v vsaki skupini isti, globalni indeksi pa so unikatni, saj se dodelijo glede na celoten indeksni prostor. Če pogledamo element z lokalnim indeksom (0,0), ki je zeleno obarvan, vidimo, da je globalni indeks (0) = 0 in globalni indeks (1) = 4, ker je v ničti vrstici in v četrtem stolpcu glede na celotni indeksni prostor.

Kadar več delovnih elementov bere in zapisuje v skupni pomnilnik, lahko pride do napake, kot je branje podatka, ki še ni bil poračunan do konca. Temu se lahko izognemo tako, da elemente sinhroniziramo. Vendar pa je sinhronizacija možna samo med delovnimi elementi iste delovne skupine znotraj izvajanja določenega ščepca. To lahko dosežemo s programskimi pregradami (angl. work-group barriers).

3.3 Pomnilniški model

Pomnilnik je v OpenCL-u razdeljen na dva dela:

- *Pomnilnik na gostitelju*: Je pomnilnik ki je direktno dostopen gostitelju. Obnašanje tega pomnilnika je definirano zunaj OpenCL-a. Pomnilniški objekti se prenašajo med gostiteljem in napravami s pomočjo funkcij, ki so definirane v OpenCL-ovem vmesniku uporabniškega programa.
- *Pomnilnik na napravi*: Je pomnilnik, do katerega lahko ščepci, ki se izvajajo na napravi, direktno dostopajo.

Delovna enota rabi pomnilnik za branje in pisanje podatkov. Vsaka delovna enota ima dostop do štirih vrst pomnilnika, ki se med seboj razlikujejo po velikosti in dostopnosti. To so:

- *Globalni pomnilnik*: Je največji in je dostopen vsem delavnim enotam.
- *Pomnilnik za konstante*: Iz njega lahko delovne enote samo berejo. Vanj lahko zapisuje samo gostitelj.
- *Lokalni pomnilnik*: Dostopen je samo delovnim enotam v isti skupini. Delovne enote zunaj te skupine ne morejo dostopati do tega pomnilnika. Uporablja se za izmenjavo podatkov v delovni skupini. Delovna enota lahko do tega pomnilnika dostopa približno 100-krat hitreje kot do globalnega pomnilnika[1].
- *Lastni pomnilnik*: Je lastni pomnilnik delovne enote in samo ta enota lahko do njega dostopa. Spremenljivke, ki so definirane v lastnem

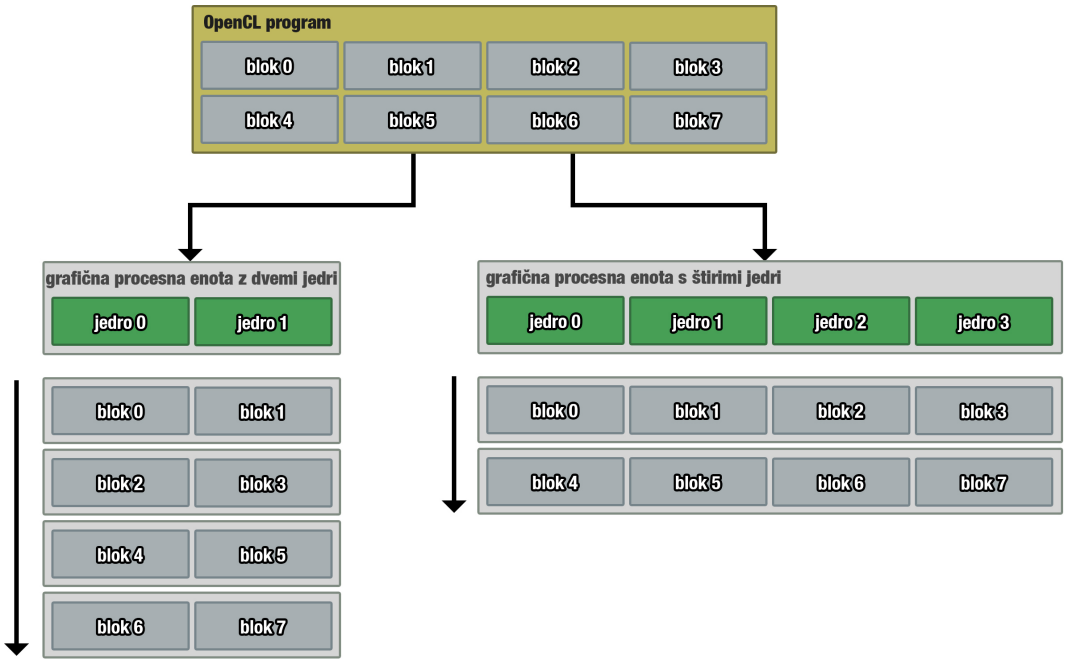
pomnilniku ene delovne enote, niso vidne drugim delovnim enotam. Dostop do tega pomnilnika je še hitrejši, kakor dostop do lokalnega pomnilnika.

Slika 3.3: Prikaz razdelitve OpenCL-ove arhitekture z vidika pomnilniškega modela. Na sliki vidimo, da imajo vsi delovni elementi dostop do globalnega pomnilnika in pomnilnika s konstantami, medtem ko lokalni pomnilnik pripada samo določeni delovni skupini in lahko do njega dostopajo samo delovni elementi tiste skupine. Lastni pomnilnik pa je dostopen samo tistemu delovnemu elementu kateremu pripada.

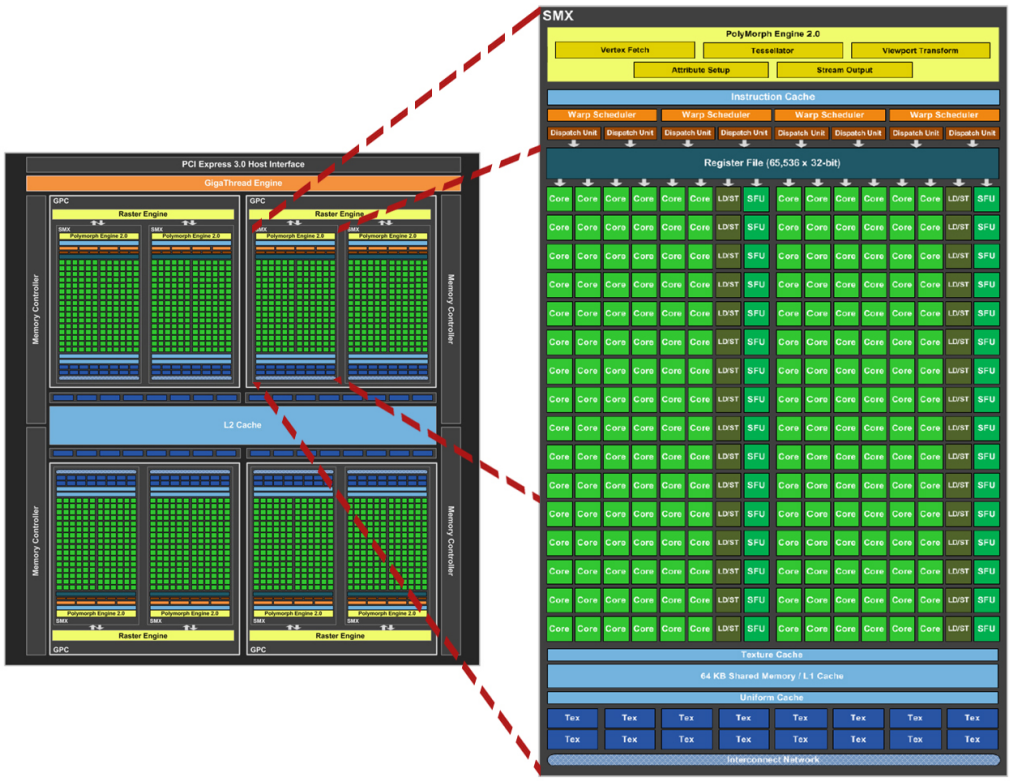
Lokalni in lastni pomnilnik sta vedno povezana z določeno napravo, medtem, ko sta globalni pomnilnik in pomnilnik za konstante deljena med vsemi napravami v danem kontekstu.[5] Kontekst je ustvarjen iz ene ali več naprav in se ga uporablja za upravljanje teh naprav.

3.4 Programski model

Zdaj pa si poglejmo kako OpenCL-ova arhitektura sovpada z grafično procesno enoto. Najnovejša Nvidia grafična kartica je razdeljena na grafično procesne skupine (angl. Graphics Processing Clusters, GPC), ki vsebujejo do štiri multiprocesorje (angl. next generation streaming multiprocessor, SMX). Enemu SMX procesorju se v izvajanje dodeli vsaj en blok niti. SMX pa je naprej razdeljen na procesne oziroma izvajalne enote, ki jih Nvidia imenuje jedra. Vsakemu jedru v SMX procesorju so dodeljene 4 niti iz bloka, ki mu je bil dodeljen. Posameznemu SMX procesorju pripada tudi skupni ali deljeni pomnilnik, preko katerega si niti v bloku lahko izmenjujejo podatke. Imamo tudi globalni pomnilnik DRAM, preko katerega si lahko podatke izmenjujejo niti iz različnih blokov. Kako izgleda arhitektura Nvidine grafične kartice GTX 680 je prikazano na sliki 3.5.



Slika 3.4: Prikaz programa, ki je razdeljen na več blokov, ki se izvedejo neodvisno eden od drugega, tako da GPE z več jedri izvede program hitreje kot GPE z manj jedri.



Slika 3.5: Arhitektura Nvidine grafične kartice GeForce GTX 680. Na sliki vidimo, da je grafična kartica razdeljena v 4 GPC-je, ki so naprej razdeljeni na 2 SMX-a. Vsak SMX pa je razdeljen na 192 jeder [9].

Poglavje 4

Implementacija

Implementacije rezanja šivov smo se lotili na dva načina. Eden je rezanje enega šiva, kar pomeni, da po vsakem odrezanem šivu, ponovno izračunamo energijo ter komulativo in na novih podatkih poiščemo šiv in ga odrežemo. Postopek ponavljamo dokler ni slika želenih dimenzij. Drug način pa je iskanje več šivov naenkrat. Pri tej metodi poiščemo več šivov, ki so primerni za odstranitev. To storimo tako, da elemente v zgornji vrstici komulative uredimo od najmanjšega do največjega. Zraven si zapomnimo tudi pripadajoče indekse. Nato vzamemo pripadajoči indeks prvega elementa v urejenem seznamu in poiščemo šiv. Med iskanjem si zapomnimo najmanjši in največji indeks v šivu. Nato vzamemo pripadajoči indeks naslednjega elementa, če je ta indeks med najmanjšim in največjim indeksom prejšnjih šivov, ga ignoriramo in se premaknemo naprej na naslednji element. To ponavljamo tako dolgo, dokler ne pridemo do zadnjega elementa v urejenem seznamu. Nato poiskane šive odrežemo in ponovno poračunamo energijo ter komulativo in postopek ponovimo. Oba načina sta implementirana za CPE in za GPE. GPE različici imata v začetku isto kodo. Iskanje platform in naprav ter združitve le-teh v kontekst je enako. Koda se začne razlikovati pri ščepcih, ko je potreben drugačen pristop pri iskanju primernega šiva in rezanju le-tega.

4.1 Priprava programa za GPE

Primarna naloga gostitelja je, da pošilja ukaze na eno ali več naprav. Program se začne na gostiteljevi strani, kjer se najprej poišče eno ali več platform z ukazom `clGetPlatformIDs`. S tem ukazom dobimo število platform in vse platforme, ki so na voljo. Nato izberemo platformo ali platforme s katerimi želimo delati. Z uporabo ukaza `clGetPlatformInfo` lahko izpišemo ime proizvajalca in ime platforme. V našem primeru je ime proizvajalca NVIDIA Corporation in ime platforme NVIDIA CUDA. S funkcijo `clGetDeviceIDs` smo poiskali vse naprave, ki so priključene na to platformo. Če rabimo informacije o napravah, uporabimo ukaz `clGetDeviceInfo`. Naša naprava, ki smo jo uporabili, ima naslednje lastnosti:

- Ime: GeForce GTX 770.
- Velikost globalnega pomnilnika: 2048 MB.
- Velikost lokalnega pomnilnika: 49151 KB.
- Število računskih enot: 8 multiprocesorjev.
- Maksimalno število delovnih enot v eni delovni skupini: 1024.
- Maksimalno število delovnih enot po dimenzijah: 1024, 1024, 64.
- Maksimalna urina frekvenca: 1137 MHz.

Ko izberemo naprave, ki jih želimo uporabiti, jih združimo v objekt tipa `cl_context`.

Nato aplikacija prebere izvorno kodo v kateri so zapisani ščepci. S to kodo potem ustvarimo objekt tipa `cl_program`, ki ga potrebujemo, da zgradimo (angl. build) program z ukazom `clBuildProgram`. Ta prevede kodo za vsako napravo v kontekstu. Potem se lahko ustvarijo objekti tipa `cl_kernels` za vsak ščepec, ki ga program vsebuje. Da gostitelj lahko komunicira z napravo, ustvari objekt tipa `cl_command_queue`. Vsak ukaz, ki se doda v ta objekt, pove ciljni napravi kaj naj izvede.

Ko pripravimo vse potrebno za komunikacijo med gostiteljem in napravo, začnemo z alokacijo pomnilnika in prenosom podatkov na grafično kartico. Za ravnanje s sliko smo uporabili knjižnico OpenCV. Najprej preberemo

sliko in jo shranimo. Na podlagi tega dobimo višino in dolžino slike, ki ju potrebujemo, da alociramo pomnilnik na grafični kartici in prenesemo podatke. To storimo z ukazom `clCreateBuffer` tako, da mu nastavimo zastavici `CL_MEM_READ_WRITE` in `CL_MEM_COPY_HOST_PTR`. Podatki se prenesejo in nastavi se dovoljenje za branje in pisanje v ta seznam. Na ta način ustvarimo tudi ostale sezname na grafični kartici za vmesne poračune kot so energija slike, komulativa slike, ter seznam za sortiranje prve vrstice komulative in pripadajoče indekse, na koncu pa še seznam za šive.

4.2 Ščepci

Ker imamo implemntirana dva načina rezanja šiva, je nekaj ščepcev skupnih, nekaj pa različnih. Najprej bomo opisali ščepce, ki so skupni, nato ščepce, ki pripadajo rezanju enega šiva in na koncu še ščepce, ki pripadajo načinu rezanja več šivov naenkrat. Ščepce ustvarimo z ukazom `clCreateKernel`, ki mu za parametre podamo objekt tipa `cl_program` in ime našega ščepca, ki ga želimo ustvariti.

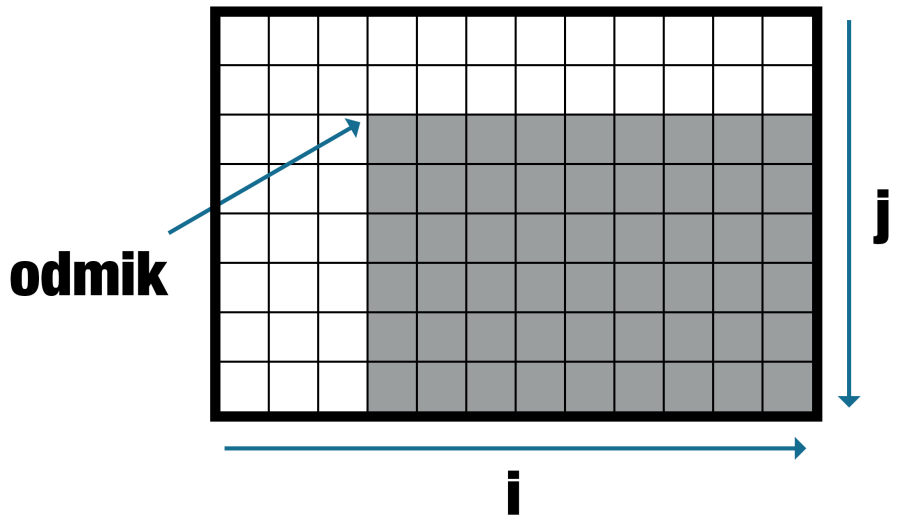
4.2.1 Skupni ščepci

Skupni ščepci:

- `SobelEnergy` : Ščepec za poračunanje energije slike. Tu je implementiran Sobelov operator, ki se izvede nad vsakim pikslom slike. Pareametri so:

  0. `global unsigned char *input`: vhodna slika,
  1. `global int *energyData`: seznam za rezultate poračunane energije,
  2. `const int width`: dolžina slike,
  3. `const int height`: višina slike,
  4. `const int controlWidth`: dolžina slike zmanjšana za že odstranjene šive.

Z ukazom `clSetKernelArg` nastavimo vsak parameter ščepca posebej. Preden zaženemo ščepec z ukazom `clEnqueueNDRangeKernel`, pa moramo določiti še nekaj parametrov. Najprej moramo določiti dimenzionalnost indeksnega prostora. Ker je slika dvodimenzionalna, nastavimo dimenzionalnost na 2. Potem določimo velikost delovne skupine, ki ima v našem primeru maksimalno velikost 1024. Na podlagi tega lahko določimo še velikost indeksnega prostora, ki mora biti večkratnik velikosti delovne skupine. Na koncu določimo še odmik, ki ga tukaj ne potrebujemo. Odmik nam pove, kje v našem indeksnem prostoru se bo računanje začelo. Ker imamo dvodimenzionalni indeksni prostor, so tudi ostali parametri zapisani kot seznam velikosti 2, tako imamo na primer velikost delovne skupine zapisane kot {32, 32}. Rezultata ne prenašamo nazaj na gostitelja, ker ga tam ne bomo rabili.



Slika 4.1: Prikaz odmika v indeksnem prostoru. Odmik je v tem primeru {3,2}. To pomeni, da se ščepec izvede samo nad delom, ki je na sliki sivo obarvan.

- `Comulative` : Ščepec za računanje komulative vrstico po vrstici. Parametri:

0. `global int *energyData`: poračunana energija slike,
1. `global int *comulativeData`: seznam za rezultate poračunane komulative,
2. `const int width`: dolžina slike,
3. `const int height`: višina slike,
4. `const int controlWidth`: dolžina slike zmanjšana za že odstranjene šive.

Ker je pri računanju komulative rezultat določene vrstice odvisen od rezultata predhodne vrstice in ker niti izven bloka ne moremo sinhronizirati, moramo komulativo poračunati vrstico za vrstico. Tako ščepec zaženemo za vsako vrstico posebej. Tu si pomagamo z odmikom. Začnemo v zadnji vrstici, zato odmik nastavimo na $\{0, v\text{-}1\}$, kjer je v enak višini slike. Ravno tako spremenimo velikost delovne skupine in indeksnega prostora, ki imata sedaj element z indeksom 1 enak 1. Ko opravimo s trenutno vrstico, odmik zmanjšamo za 1 in ponovno zaženemo ščepec, to ponavljamo dokler ne pridemo do prve vrstice. Ta ščepec se tako izvede (v-1)-krat.

- `ComulativeKElements` : Ščepec za računanje komulative vsake k-te vrstice, kjer je k večji od 0 in manjši od v. To pomeni, da se premikamo za korak k proti zgornji vrstici. Parametri:

  0. `global int *energyData`: poračunana energija slike,
  1. `global int *comulativeData`: seznam za rezultate poračunane komulative,
  2. `const int controlWidth`: dolžina slike zmanjšana za že odstranjene šive,
  3. `const int width`: dolžina slike,
  4. `const int height`: višina slike,
  5. `const int step`: korak med vrsticami oziroma k.

  Ker bi radi pohitrili računanje komulative, smo se odločili poračunati samo vsako k-to vrstico. Še vedno moramo paziti, da poračunamo vse

vrstice in, da so poračunane ena za drugo, zato v kodi postavimo pregrade, ki zagotovijo zaporedno izvajanje. Pojavi se problem pri robnih elementih. Za elemente, ki pripadajo drugemu bloku, ne vemo ali so že poračunani ali ne. Zato jih mora blok poračunati sam. To pomeni, da se robni primeri poračunajo večkrat. V naslednjem poglavju, kjer bodo rezultati meritev, bomo videli ali izvajanje ščepca vsako k-to vrstico odtehta večkratno poračunavanje robnih primerov. Tudi tukaj moramo spremeniti velikost delovne skupine in velikost indeksnega prostora. V obeh primerih je sedaj element z indeksom 1 enak k-ju, odmik pa je $\{0, v\text{-}\ k\}$. Velikost koraka je k, zato se ščepec kliče v/k krat, če je k večkratnik v-ja in $(v/k{+}1)$ krat, če ni.

4.2.2 Rezanje enega šiva

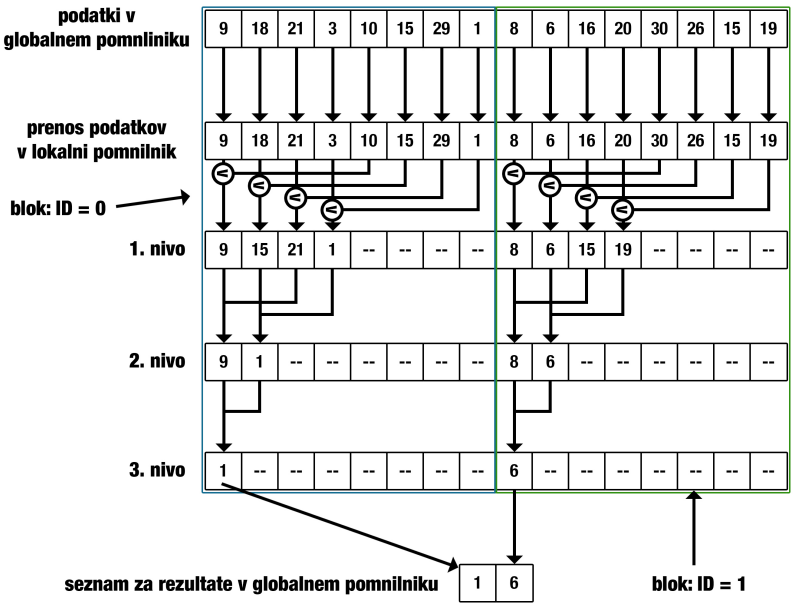
Ščepci rezanja enega šiva:

- `Reduce` : Ščepec, s katerim v prvi vrstici komulative poiščemo najmanjši element vsakega bloka in njim pripadajoče indekse. Parametri:

  0. `global int* comulative`: poračunana komulativa,
  1. `local int* scratch`: lokalni seznam nad katerim izvajamo redukcijo,
  2. `local int* scratchIndex`: seznam pripadajočih indeksov zgornjemu seznamu,
  3. `const int controlWidth`: dolžina slike zmanjšana za že odstranjene šive,
  4. `global int* result`: seznam za rezultate redukcije,
  5. `global int* resultIndex`: seznam za pripadajoče indekse rezultatom redukcije.

  S pomočjo redukcije poiščemo najmanjše elemente blokov. Ker redukcijo izvedemo samo na prvi vrstici poračunane komulative, je element z indeksom 1 pri velikosti delovne skupine in indeksnem prostoru zopet enak 1. Odmika tokrat ne potrebujemo. Tukaj domnevamo, da je

velikost bloka potenca števila dva. Ker seveda ni rečeno, da je dolžina slike potenca števila dva, zapolnimo manjkajoče elemente z veliko vrednostjo (`INFINITY`) in mu pripadajoči indeks nastavimo na -1. Vsak blok poišče minimalni element v svojem prostoru in nato ga element z lokalnim indeksom 0 zapiše v seznam `result`, na mesto z indeksom, ki je enak indeksu bloka. V našem primeru je redukcija implementirana tako, da najprej vzamemo polovico dolžine seznama in jo shranimo v spremenljivko `offset`. Nato si vsaka nit shrani svoj lokalni indeks v spremenljivko `local_id`. Če je `local_id` niti manjši od `offset`ta, se primerjata elementa z indeksom `local_id` in `local_id` povečan za `offset`. Tisti, ki je manjši se zapiše na mesto z indeksom `local_id`. Nato `offset` delimo z 2 in postopek ponovimo. To delamo dokler je `offset` večji od 0. Postopek je prikazan na sliki 4.2.



Slika 4.2: Prikaz redukcije.

- `ReductionComplete` : Ščepec, s katerim zaključimo redukcijo in na podlagi rezultata poiščemo šiv. Parametri:

  0. `global int* comulative`: poračunana komulativa,
  1. `global int* resultMin`: rezultati prejšnje redukcije,
  2. `global int* resultIndexMin`: pripadajoči indeksi rezultatom redukcije,
  3. `local int* scratch`: lokalni seznam nad katerim izvajamo redukcijo,
  4. `global local int* scratchIndex`: seznam pripadajočih indeksov zgornjemu seznamu,
  5. `const int resultWidth`: velikost seznama `resultMin`,
  6. `const int width`: dolžina slike,
  7. `const int height`: višina slike,
  8. `global int* seam`: seznam za shranitev poračunanega šiva.

  Za izvedbo tega ščepca odmika in lokalne velikosti ne nastavimo. Dimenzionalnost nastavimo na 1, ter globalno velikost na najmanjšo možno potenco števila dva, ki je še večja od velikosti seznama z rezultati redukcije. To potrebujemo zato, da lahko najprej opravimo še redukcijo nad rezultatom prejšnjega ščepca. Potem pa na podlagi dobljenega rezultata nit z indeksom 0 poišče šiv. Ker iskanje šiva ne moremo paralelizirati in ker prenašanje podatkov iz GPE na CPE stane, smo se odločili iskanje šiva opraviti kar na GPE.

- `CutOneSeam` : Ščepec, v katerem odrežemo šiv, ki smo ga našli v prejšnjem ščepcu. Parametri:

  0. `global unsigned char *data`: podatki vhodne slike,
  1. `global unsigned char *dataCopy`: kopija podatkov vhodne slike,
  2. `global int *seam`: šiv, ki ga moramo izrezati,
  3. `const int controlWidth`: dolžina slike zmanjšana za že odstranjene šive,

4. `const int width`: dolžina slike,
5. `const int height`: višina slike.

Šiv odstranimo tako, da piksle, ki so pred šivom samo prepišemo, piksle za šivom pa prestavimo za eno mesto v levo. Tukaj bi morali zagotoviti, da ne preberemo podatka, ki še ni bil obdelan. Ker sinhronizacija med bloki ni mogoča, rešimo problem tako, da imamo na GPE dva seznama, prvega za branje podatkov, drugega za zapisovanje. Dimenzionalnost nastavimo nazaj na 2. Odmika ne potrebujemo. Velikost delovne skupine nastavimo na poljubno številko, ki je manjša ali enaka 1024. Nastavimo tudi velikost indeksnega prostora, ki mora biti večkratnik velikosti delovne skupine in je hkrati večja od vhodne slike, da zajamemo celo sliko.

4.2.3 Rezanje več šivov naenkrat

Ščepci v primeru rezanja več šivov naenkrat:

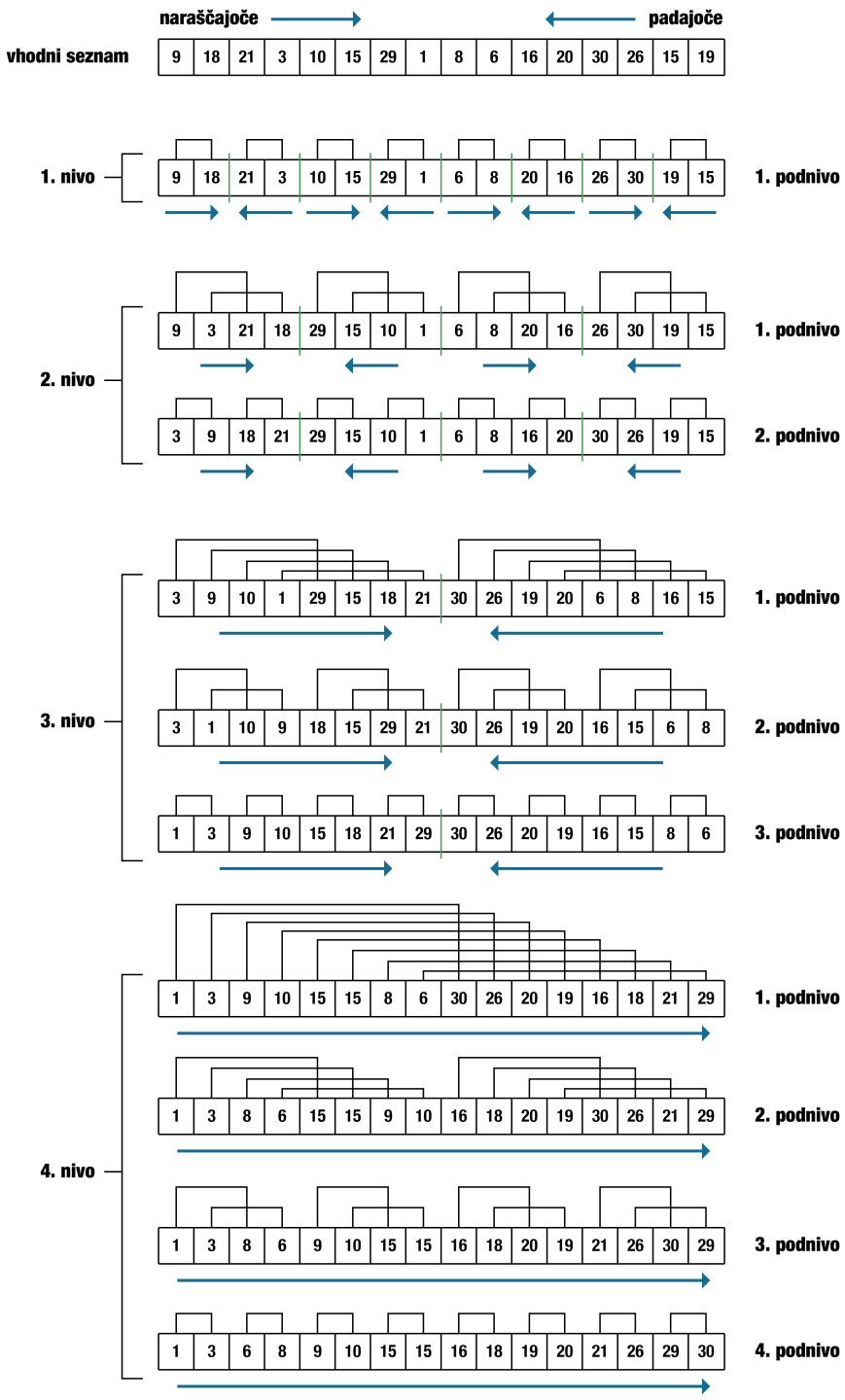
- `BitonicSort` : Ščepec, v katerem implementiramo bitonično urejanje podatkov. To storimo nad podatki prve vrstice komulative. Parametri:

  0. `global int * data`: podatki ki jih hočemo sortirati,
  1. `global int * comulativeIndexes`: pripadajoči indeksi podatkov za urejanje,
  2. `const uint stage`: nivo urejanja,
  3. `const uint subStage`: podnivo urejanja,
  4. `const uint direction`: smer urejanja, naraščajoče ali padajoče.

  Bitonično zaporedje sestavljata dve monotoni zaporedji, naraščajoče in padajoče zaporedje. Bitonično urejanje lahko uporabljamo samo na bitoničnih zaporedjih. Se pravi, če hočemo bitonično urejanje uporabiti na poljubnem seznamu, ga moramo najprej pretvoriti v bitonično zaporedje. To naredimo tako, da postopoma združujemo vedno večje

dele seznama v bitonična zaporedja. Bitonično urejanje je prikazano na sliki 4.3.

Najprej kopiramo elemente prve vrstice komulative, da lahko nad njo opravimo urejanje ne da bi izgubili originalno zaporedje rezultatov komulative, ki jih rabimo kasneje za iskanje šivov. Nato seznam uredimo z bitoničnim urejanjem. Tudi tukaj mora biti seznam velikosti potence števila dva, zato manjkajoče elemente zapolnimo z `INFINITY`. Ta potenca nam pove koliko nivojev bomo imeli. Če je naš seznam velik 2^n in je n naravno število, potem je število nivojev enako n. Vsak nivo k pa ima k podnivojev. Se pravi, če smo na 3. nivoju imamo 3 podnivoje. Na CPE imamo tako dve zanki, eno glavno za nivoje in eno vgnezdeno za podnivoje. V vsakem podnivoju zaženemo naš ščepec. Odmika tukaj ne potrebujemo, saj moramo zajeti vse elemente. Dimenzionalnost nastavimo na 1. Velikost indeksnega prostora je polovica velikosti seznama za urejanje. Velikost delovne skupine je ravno tako velikost potence števila dva, ki je manjše ali enako naši maksimalni velikosti 1024.



Slika 4.3: Prikaz bitoničnega urejanja. Slika prikazuje, kako skozi nivoje in podnivoje gradimo bitonična zaporedja, dokler na koncu nimamo samo eno monotono zaporedje.

- `FindSeams` : V tem ščepcu poiščemo vse primerne šive za odstranitev. Parametri so:

    0. `global int *comulativeIndexes`: urejeni indeksi prve vrste komulative,
    1. `global int *comulative`: celoten seznam komulativ,
    2. `global int *seams`: seznam za shranjevanje šivov,
    3. `local int2 *limitArray`: lokalni seznam za shranjevanje najmanjšega in največjega indeksa v vsakem šivu,
    4. `local int *tmpSeam`: seznam, ki ga polnimo medtem ko iščemo šiv,
    5. `local int *seamCopy`: seznam za pomoč pri urejanju vrstice pri seznamu `seams`,
    6. `const int numberOfSeams`: število šivov,
    7. `const int height`: višina slike,
    8. `const int width`: dolžina slike,
    9. `global int *seamCounter`: nam pove koliko šivov smo dejansko našli.

    V tem ščepcu se po vrsti sprehajamo po seznamu `comulativeIndexes`. Najprej vzamemo vrednost na ničtem indeksu, ki vsebuje indeks najmanjšega elementa v prvi vrstici komulative. Potem poiščemo šiv in ga kopiramo v seznam `seams`, kjer hranimo vse primerne šive. Med iskanjem si zapomnimo minimalni in maksimalni indeks šiva, ter ga shranimo v seznam `limitArray`. Ko iščemo naslednji šiv, najprej pogledamo, če je v mejah prejšnjih šivov. Če je, ga ignoriramo, drugače pa kopiramo v seznam. Da nam bo pri rezanju teh šivov lažje, si pomagamo tako, da šive, ki so v svoji višini uredimo. Tako bomo pri rezanju točno vedeli za koliko mest določen piksel prestavimo. To pomeni, da se sprehodimo čez seznam indeksov, ki jih moramo odstraniti v določeni vrstici, in ko naletimo na indeks, ki je večji od našega se ustavimo. Kolikor je manjših elementov, za toliko mest trenutni piksel

premaknemo v levo. Dimenzionalnosti ne spreminjamo, ravno tako ne odmika. Ker iskanja šiva ne moremo paralelizirati, velikost indeksnega prostora in velikost delovne skupine nastavimo na 1.

- `CutSeams` :

  0. `global unsigned char *data`: podatki vhodne slike,
  1. `global int *seams`: seznam šivov, ki jih želimo odstraniti,
  2. `const int controlWidth`: dolžina slike zmanjšana za že odstranjene šive,
  3. `const int width`: začetna dolžina slike,
  4. `const int height`: višina slike,
  5. `const int numberOfSeams`: število šivov, ki jih želimo porezati.

  V tem ščepcu porežemo šive, ki smo jih dobili v prejšnjem ščepcu. To storimo tako, da se vsaka nit sprehodi čez seznam indeksov, ki jih je v njeni vrstici potrebno odstraniti. Z elementi seznama primerja svoj globalni indeks (0). Kolikor je v tem seznamu manjših elementov, za toliko mest se prestavi piksel, ki ji pripada. Tudi tukaj moramo zagotoviti, da ne preberemo podatek, ki še ni bil obdelan, zato imamo na GPE dva seznama, eden za branje podatkov, drugi za zapisovanje. Dimenzionalnost lahko zopet nastavimo nazaj na 2, ter delovno skupino na velikost 1024 in indeksni prostor na večkratnik delovne skupine.

Poglavje 5

Rezultati meritev

Pogledali si bomo rezultate, ki smo jih dobili pri meritvi časovnega izvajanja posameznih ščepcev in navadnih funkcij. Videli bomo, kako določitev velikosti bloka vpliva na izvedbo ščepca. Za ščepce imamo tri načine meritve:

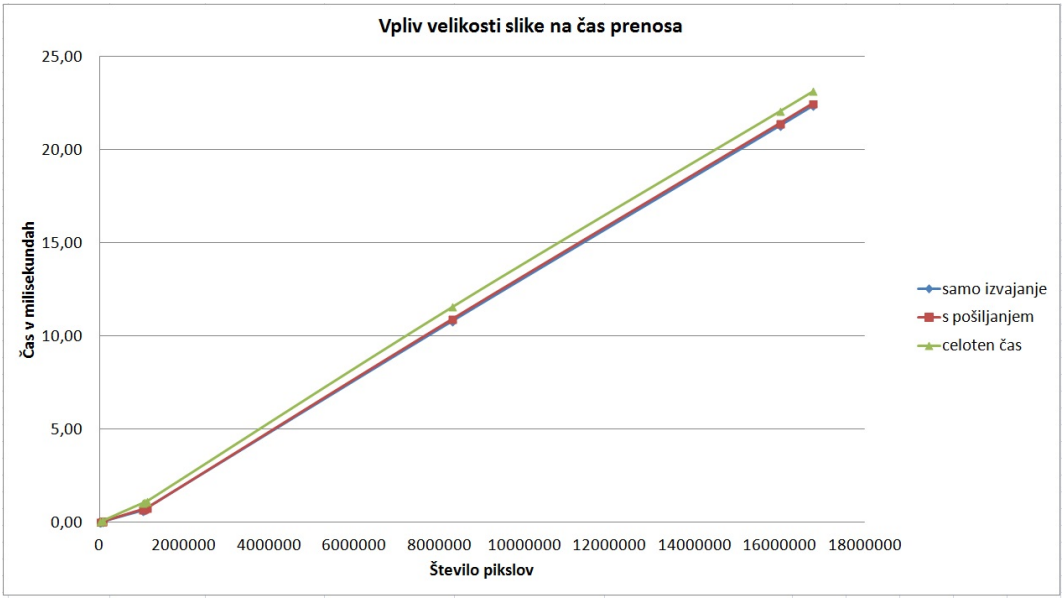
- izvajane: samo izvajanje ščepca,
- pošiljanje: poleg izvajanja ščepca upoštevamo še pošiljanje ukaza na GPE,
- ukazna vrsta: še dodatno upoštevamo dodajanje ukaza v ukazno vrsto.

Pri primerjavi med CPE in GPE smo vzeli zadnjo, saj nam ta pove celoten čas izvajanja za nek ščepec. Na koncu pa si bomo pogledali še koliko časa se obe različici programa izvajata na CPE in na GPE.

5.1 Prenos podatkov na GPE

Izmerili smo vpliv velikosti slike na prenos podatkov iz CPE na GPE. Najmanjša slika, ki smo jo prenesli na GPE je imela dimenzije 100x100. Za to sliko smo izmerili čas izvedbe, ki je 0,004 ms, čas izvedbe s pošiljanjem na napravo je 0,02 ms. Celoten čas prenosa podatkov, od dodajanja ukaza v ukazno vrsto do izvedbe, pa je trajal 0,03 ms. Največja slika s katero smo testirali prenos pa ima dimenzije 4096x4096. Pri tej je bil celoten čas izva-

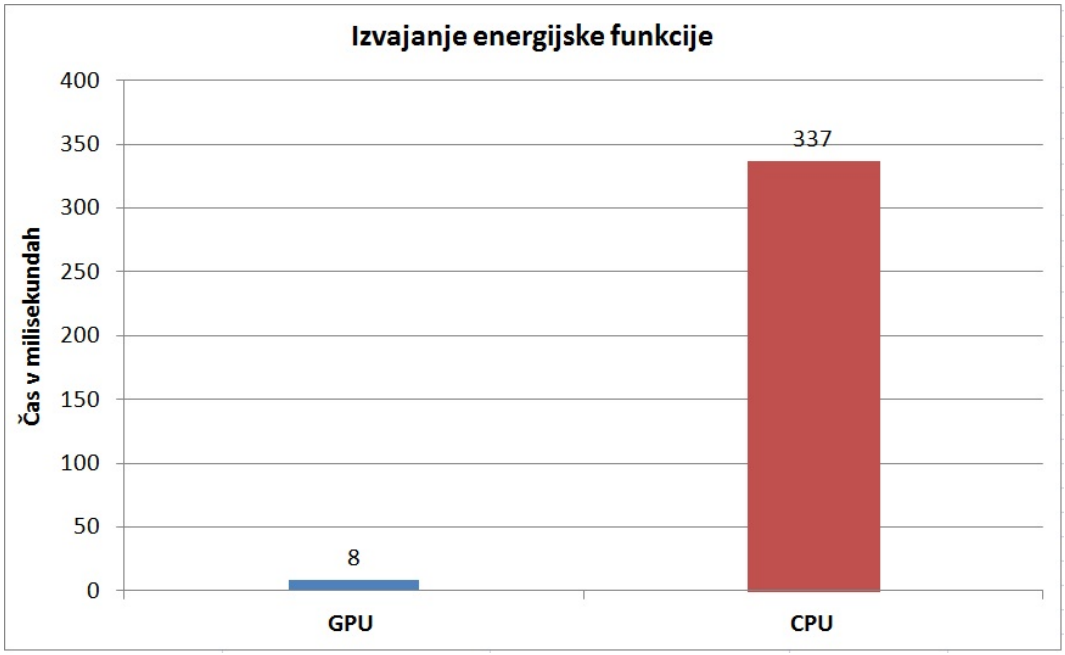
janja 23 ms, čas s pošiljanjem 22 ms in sam čas izvajanja 22 ms. Slika 5.1 prikazuje razmerje med temi tremi časi.



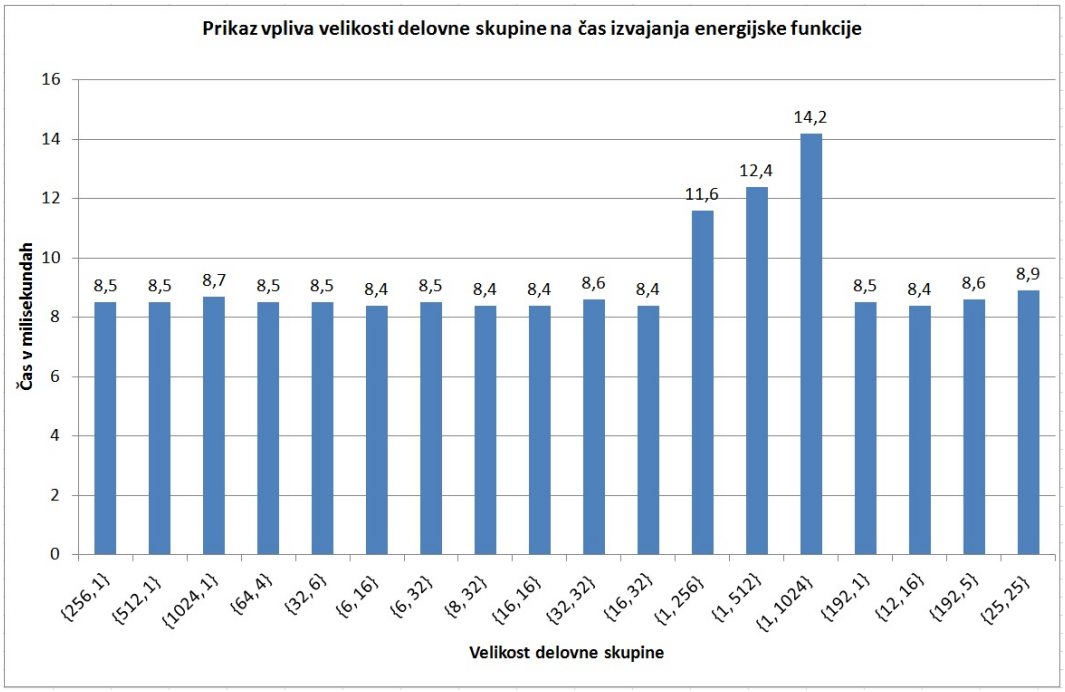
Slika 5.1: Graf prikazuje čas prenosa podatkov iz CPE na GPE.

5.2 Poračun energije

Vse meritve od tukaj dalje smo izvajali na sliki velikosti 1280x868. Pri tem ščepcu smo najprej pogledali, kako velikost delovne skupine vpliva na časovno izvedbo. Rezultati merjenja so prikazani na sliki 5.3. Ugotovili smo, da je velikost 8x32 najbolj optimalna velikost delovne skupine. Čas izvedbe ščepca, smo vzeli za primerjavo z izvedbo energijske funkcije na CPE. Ugotovili smo, da je izvedba ščepca na GPE kar 97% hitrejša od izvedbe energijske funkcije na CPE, saj izvajanje na GPE traja 8 ms, na CPE pa kar 337 ms. Pri večini drugih ščepcev, kjer je indeksni prostor dvo-dimenzionalen, se izkaže, da je najbolj optimalna velikost delovne skupine 16x16. Možna izboljšava bi bila prenos podatkov iz glavnega pomnilnika v lokalni pomnilnik, pred dejanskim računanjem energije piksla. S tem bi si omogočili manj dostopanj do globalnega pomnilnika.



Slika 5.2: Graf prikazuje čas izvajanja energijske funkcije na CPE in na GPE.



Slika 5.3: Graf prikazuje čase izvajanja ščepca za računanje energije na GPE, z različnimi velikostmi delovne skupine.

5.3 Poračun komulative

Za poračun komulative eno vrstico za drugo smo vzeli velikost delovne skupine 1024x1, ki je po naših meritvah najbolj optimalna. Čas izvajanja za celotno računanje komulative je 34 ms. Ker je višina slike velika 686 vrstic, se ščepec za računanje komulative izvede 686-krat. Če od celotne meritve odštejemo meritve za samo izvajanje tega ščepca, ki je 16 ms, ugotovimo, da nas dodajanje ščepca v ukazno vrsto in pošiljanje na GPE, v povprečju stane 18 ms.

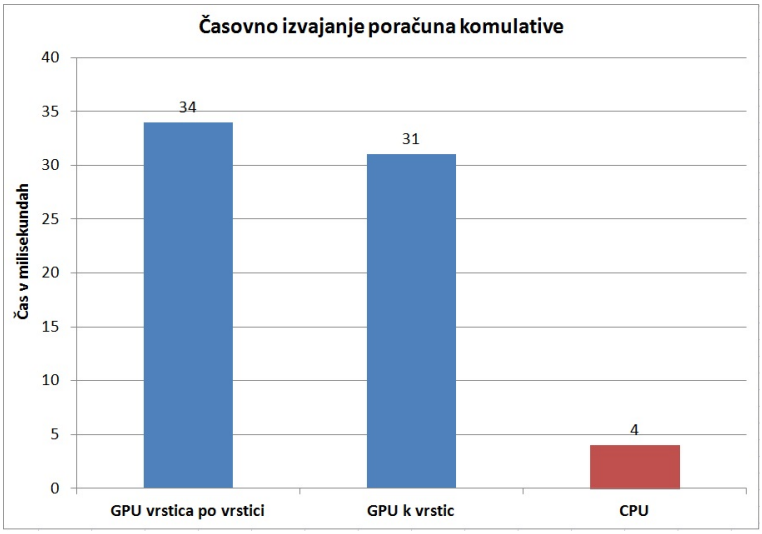
Pri testiranju časa izvajanja komulative vsako k-to vrstico, pa smo ugotovili, da je zanka *for* precej zahtevna za izvajanje na grafični kartici. Za dodatno testiranje, smo v zanki *for* pobrisali vse, razen prepis energijske vrednosti piksla v komulativo. S tem smo se znebili notranje zanke, ki računa

robne piksle iz sosednjega bloka. Rezultati so prikazani v tabeli 5.1. Pohitritev ščepca bi verjetno dosegli že s tem, da piksle iz sosednjega bloka poračunamo samo enkrat na začetku. Vse to nas privede do razmišljanja, da bi fiksno določili korak k in se čisto znebili zank *for* v našem ščepcu.

Čas izvajanja na CPE je 4 ms in je bistveno hitrejši od izvajanja na GPE.

	k			
ščepec	1	2	3	4
dejansko računanje komulative	39	31	32	35
samo prepis energijske vrednosti	37	22	17	15

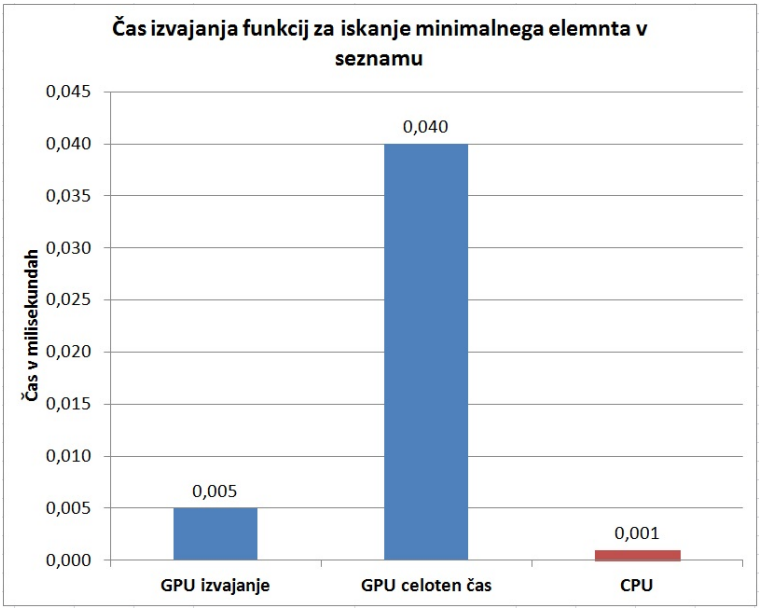
Tabela 5.1: Prikaz izvajanja ščepca za računanje k vrstic naenkrat. Časi so v milisekundah.



Slika 5.4: Graf prikazuje čas izvedbe funkcije za računanje komulative na GPE in CPE. Na tem grafu je k enak 2.

5.4 Redukcija

Iskanje minimalnega elementa v zgornji vrstici komulative vzame na CPE 0,001 ms. Na GPE smo uporabili redukcijo za iskanje minimalnega elementa. Celotna časovna izvedba redukcije na seznamu velikosti 1280 je trajala 0,04 ms. Če ne upoštevamo časa za dodajanje v ukazno vrsto in časa za pošiljanje na grafično kartico, čas izvedbe samega ščepca traja 0,005. V našem primeru je velikost seznama premajhna, da bi odtehtala izvedbo ščepca na GPE.

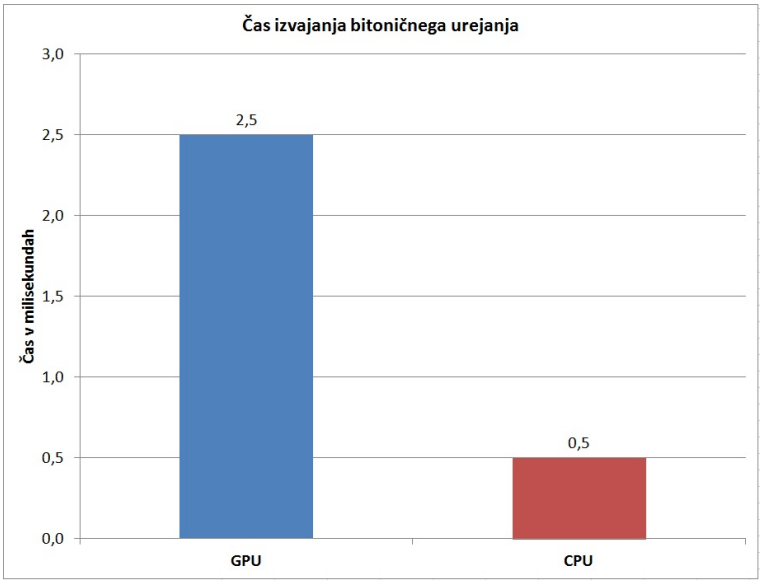


Slika 5.5: Graf prikazuje čas izvedbe funkcij za iskanje minimalnega elemnta na GPE in CPE.

5.5 Bitonično urejanje

Pri pregledovanju literature, ki smo jo uporabili za izdelavo diplomske naloge, smo ugotovili, da se za urejanje na GPE največkrat pojavljata korensko urejanje (angl. radix sort) in bitonično urejanje. Odločili smo se za implementacijo slednjega.

Bitonično urejanje na CPE traja 0,5 ms, na GPE pa kar 2,5 ms. Pri urejanju seznama velikosti 1280 z bitoničnim urejanjem se ščepec izvede 66-krat. V povprečju se en ščepec izvede v 0,0378 ms. Ali bi se izvajanje ščepca v primerjavi z bitoničnem urejanjem na CPE pohitrilo ob večjem številu elementov v času te naloge nismo testirali.



Slika 5.6: Graf prikazuje čas izvedbe funkcij za bitonično urejanje na GPE in CPE.

5.6 Iskanje šiva

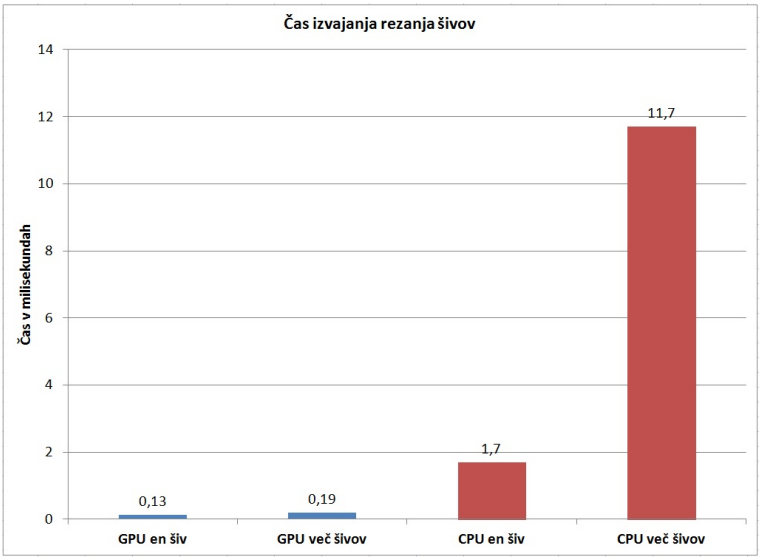
Na CPE iskanje enega šiva traja 0,009 ms, iskanje več šivov naenkrat pa 3,8 ms, kar je pričakovani rezultat. Na GPE pa traja iskanje enega šiva 0,5 ms, iskanje več šivov pa kar 442 ms. Poraja se vprašanje, če ne bi bilo bolj optimalno prenesti podatke nazaj na CPE in tam poiskati šiv oziroma šive. Iskanje več šivov na enkrat smo testirali s 4 šivi.

5.7 Prenos podatkov nazaj na CPE

Branje ene številke, kot je na primer koliko šivov je bilo najdenih, se izvaja 0,02 ms. Medtem ko branje slike velikosti 1280x868 traja 1,2 ms. Kar je blizu prenosa iste velikosti podatkov na GPE. To nam pove, da bi verjetno dobili hitrejše izvajanje programa pri rezanju več šivov, če bi podatke prenesli nazaj na CPE, tu poiskali šive in jih poslali nazaj na GPE.

5.8 Rezanje šiva

Rezanje enega šiva na CPE se izvaja 1,7 ms, rezanje več šivov pa 11,7 ms. Na GPE pa rezanje enega šiva ob izbiri velikosti delovne skupine 16x16 traja 0,13 ms, rezanje več šivov naenkrat pa 0,19 ms. V tem primeru je izvajanje rezanja šivov boljše na GPE, kot pa na CPE.



Slika 5.7: Graf prikazuje čas izvedbe funkcij za rezanje šiva na GPE in CPE.

5.9 Primerjava skupnih rezultatov

Najprej si poglejmo rezultate meritev rezanja enega šiva. Pri merjenju smo vzeli sliko dimenzij 1280x868 ter poiskali in odrezali 600 šivov. Pri skupnem merjenju programa smo ugotovili, da je različica za GPE hitrejša za 92%. Čas merjenja na GPE znaša 12 s, na CPE pa 167 s. Če si pogledamo posamezne meritve funkcij in ščepcev za rezanje enega šiva, ki so prikazani v tabeli 5.3, vidimo, da je skupen čas ščepcev, ki se izvedejo na GPE 42,16 ms in skupen čas funkcij, ki se izvedejo na CPE 342,71 ms. Vidimo tudi, da je razlog v počasnem izvajanju na CPE računanje energije vhodne slike.

GPE: en šiv	
prenos podatkov	1,1
poračun energije	8
poračun komulative	31
redukcija	0,04
iskanje šiva	0,5
kopiranje podatkov	0,19
rezanje šivov	0,13
branje rezultatov	1,2
SKUPAJ	42,16

CPE: en šiv	
prenos podatkov	/
poračun energije	337
poračun komulative	4
iskanje minimuma	0,001
iskanje šivov	0,009
kopiranje podatkov	/
rezanje šivov	1,7
branje rezultatov	/
SKUPAJ	3432,71

Tabela 5.2: Tabela prikazuje seznam vseh funkcij pri iskanju enega šiva, ki se kličejo na CPE, ter seznam ščepcev, ki se kličejo na GPE. Časi so v milisekundah.

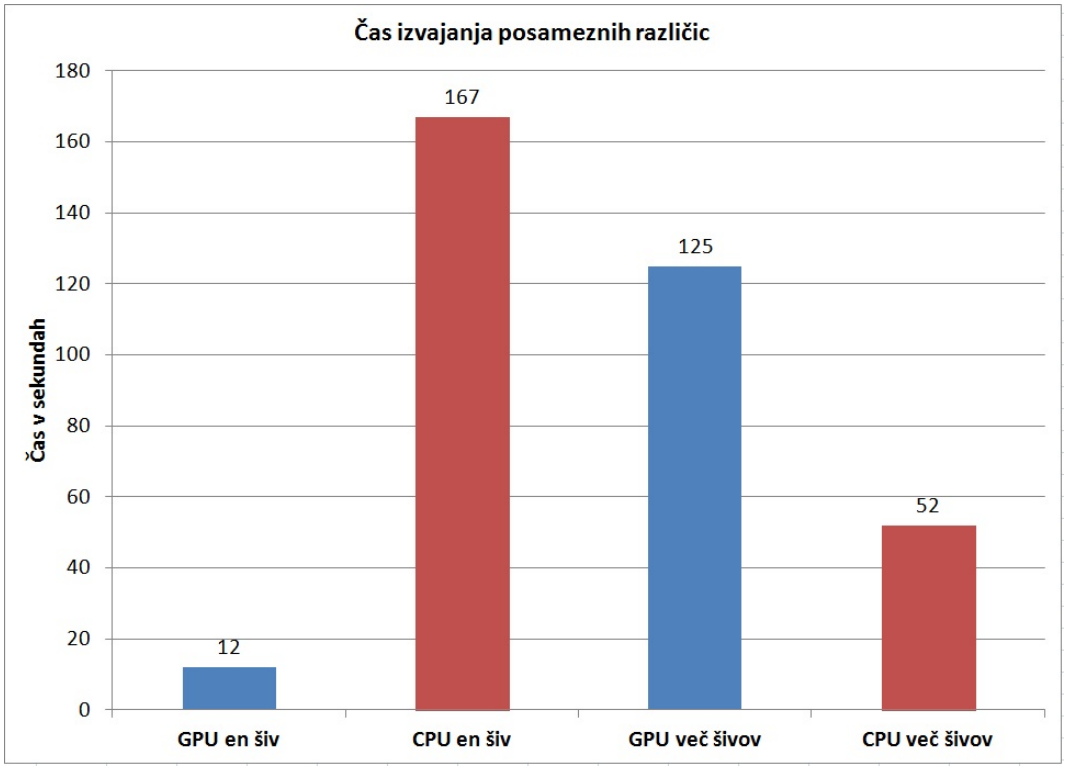
Sedaj pa si poglejmo rezultate rezanja več šivov na enkrat. V tabeli 5.4 lahko vidimo kateri ščepci se kličejo na GPE in katere funkcije se kličejo na CPE. Zraven so tudi pripadajoči rezultati meritev, ter skupen seštevek teh meritev. Tukaj naj še enkrat spomnimo, da so meritve narejene ob iskanju in rezanju 4 šivov naenkrat. Kot vidimo, je skupen čas izvajanja na GPE 486,24 ms in je bistveno večji od skupnega časa izvajanja na CPE, ki znaša 357,002 ms. Odločili smo se, da rezultat preverimo še s skupnim merjenjem

izvajanja programa. Tudi tu smo to storili s sliko, ki ima dimenzije 1280x868, na kateri smo poiskali in odrezali 600 šivov. Čas izvajanja na GPE je 125 s in na CPE 52 s. To pomeni, da se program na CPE izvede za 58% hitreje kot program na GPE. Takih rezultatov seveda nismo pričakovali. Kot vidimo v tabeli, sta največji problem ščepec za računanje komulative in pa ščepec za iskanje šivov.

GPE: več šivov naenkrat	
prenos podatkov	1,1
poračun energije	8
poračun komulative	31
kopiranje prve vrstice	0,04
bitonično sortiranje	2,5
iskanje šivov	442
branje podatka	0,02
kopiranje podatkov	0,19
rezanje šivov	0,19
branje rezultatov	1,2
SKUPAJ	486,24

CPE: več šivov naenkrat	
prenos podatkov	/
poračun energije	337
poračun komulative	4
kopiranje prve vrstice	0,002
bitonično sortiranje	0,5
iskanje šivov	3,8
branje podatka	/
kopiranje podatkov	/
rezanje šivov	11,7
branje rezultatov	/
SKUPAJ	357,002

Tabela 5.3: Tabela prikazuje seznam vseh funkcij pri iskanju več šivov naenkrat, ki se kličejo na CPE, ter seznam ščepcev, ki se kličejo na GPE. Časi so v milisekundah.



Slika 5.8: Graf prikazuje čas izvajanja posameznih različic.

Poglavje 6

Sklepne ugotovitve

Na podlagi meritev smo ugotovili, da je implementacija algoritma za rezanje enega šiva na GPE smiselna, saj se izvede kar za 92% hitreje, kot različica za CPE. Če pregledamo posamezne ščepce vidimo, da nam največji problem dela ščepec za računanje komulative. Ščepec bi lahko pohitrili, če bi ga implementirali tako, da si na začetku prepiše vrednosti iz globalnega pomnilnika v lokalni pomnilnik. Vendar pa bi še vedno ostalo večkratno klicanje ščepca, kar pomeni kar nekaj dodatnega časa za dodajanje v ukazno vrsto in pošiljanje na GPE. Ta čas bi lahko zmanjšali z izvajanjem komulative vsako k-to vrstico. Pri pregledu rezultatov med tema dvema ščepcema ugotovimo, da se računanje malo pohitri, vendar bi lahko dosegli še večjo pohitritev z optimizacijo samega ščepca. Znebili bi se notranje zanke *for*, tako da bi robne primere poračunali samo enkrat na začetku izvedbe ščepca. Možna pohitritev bi bila lahko tudi fiksna določitev vrednosti k-ja, s čimer bi se čisto znebili zank. Tudi funkcija za iskanje minimalnega elementa je bila slabša kakor na CPE. To pripisujemo dejstvu, da se je redukcija izvedla na manjšem številu elementov in tako ni prišla do izraza.

Pri implementaciji algoritma na način, kjer odrežemo več šivov naenkrat pa smo naleteli na kar nekaj presenečenj. Meritve so pokazale, da je program, ki se izvaja na GPE počasnejši. Po pregledu ščepcev, smo ugotovili, da nam problem dela ščepec za iskanje šivov. V kodi tega ščepca imamo zanko *while*,

ki se sprehaja po urejeni zgornji vrstici podatkov dokler ne pride do konca. Kar je očitno zelo zahtevno za izvedbo na GPE. Možni izboljšavi bi bili drugačna oziramo boljša implementacija samega ščepca ali prenos urejenih podatkov na CPE in iskanje šivov na gostitelju, ter potem prenos najdenih šivov nazaj na GPE. Na ta način bi pohitrili izvajanje te različice na GPE.

Ugotovili smo, da na uspešno pohitritev programa na GPE vpliva kar nekaj dejavnikov. Najbolj pomembna izmed njih je uspešna implementacija ščepcev. Videli smo da zanki *while* in *for* slabo vplivata na izvajanje ščepcev, zato se jim je najbolje izogniti. Poleg tega je priporočljiva uporaba lokalnega pomnilnika v kolikor je to mogoče.

Literatura

[1] M. Scarpino, "OpenCL in Action", Manning Publications, 2012

[2] R. Tay, "OpenCL Parallel Programming Development Cookbook", Packt Publishing, 2013

[3] S. Avidan, A. Shamir, "Seam Carving for Content-Aware Image Resizing", *ACM Transactions on Graphics*, vol. 26, no. 3, 2007.

[4] J. Tompson, K. Schlachter. "An Introduction to the OpenCL Programming Model", 2012. [Online]. Dosegljivo: http://www.cs.nyu.edu/ lerner/spring12/Preso07-OpenCL.pdf. [Dostopano 25. 1. 2016].

[5] Khronos OpenCL Working Group "The OpenCL Specification", 2015. [Online]. Dosegljivo:https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf. [Dostopano 26. 1. 2016].

[6] R. Češnovar "Rezanje šivov na grafičnih procesnih enotah z arhitekturo CUDA", 2010. [Online]. Dosegljivo: http://eprints.fri.uni-lj.si/1254/. [Dostopano 27. 2. 2016].

[7] Broadway tower. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/Seam_carving. [Dostopano 27. 2. 2016].

[8] Kingfisher. [Online]. Dosegljivo: http://image4ar.com/image/23/cms-creative-164657191-kingfisher. [Dostopano 27. 2. 2016].

[9] Nvidia. [Online]. Dosegljivo: http://www.hardwarezone.com.sg/product-nvidia-geforce-gtx-680-reference-card. [Dostopano 27. 2. 2016].